调查 RAM 使用情况

开发 Android 应用时，请始终注意您的应用使用了多少随机存取存储器 (RAM)。尽管 Dalvik 和 ART 运行时会执行例行垃圾回收 (GC)，您仍然需要了解应用何时以及在哪里分配和释放内存。为了提供稳定的用户体验，使 Android 操作系统能够在应用之间快速切换，请确保您的应用在用户不与其交互时不会消耗不必要的内存。

即使您在开发过程中遵循了管理应用的内存的所有最佳做法，您仍然可能泄漏对象或引入其他内存错误。唯一能够确定您的应用尽可能少地使用内存的方法是，利用本文介绍的工具分析应用的内存使用情况。

解读日志消息

开始调查您的应用内存使用情况的最简单切入点是运行时日志消息。有时，发生垃圾回收时，您可以在 logcat 中查看消息。

Dalvik 日志消息

在 Dalvik（而不是 ART）中，每次垃圾回收都会将以下信息打印到 logcat 中：

```
D/dalvikvm: <GC_Reason> <Amount_freed>, <Heap_stats>,
<External_memory_stats>, <Pause_time>

示例：
D/dalvikvm( 9050): GC_CONCURRENT freed 2049K, 65% free 3571K/9991K, external
4703K/5261K, paused 2ms+2ms
```

- 垃圾回收原因<GC_Reason>GC_CONCURRENT
  什么触发了垃圾回收以及是哪种回收。可能出现的原因包括：
  - GC_CONCURRENT
    在您的堆开始占用内存时可以释放内存的并发垃圾回收。
  - GC_FOR_MALLOC
    堆已满而系统不得不停止您的应用并回收内存时，您的应用尝试分配内存而引起的垃圾回收。
  - GC_HPROF_DUMP_HEAP
    当您请求创建 HPROF 文件来分析堆时出现的垃圾回收。
  - GC_EXPLICIT
    显式垃圾回收，例如当您调用 gc() 时（您应避免调用，而应信任垃圾回收会根据需要运行）。
  - GC_EXTERNAL_ALLOC
    这仅适用于 API 级别 10 及更低级别（更新版本会在 Dalvik 堆中分配任何内存）。外部分配内存的垃圾回收（例如存储在原生内存或 NIO 字节缓冲区中的像素数据）。
- 释放量<Amount_freed>freed 2049K
  从此次垃圾回收中回收的内存量。
- 堆统计数据<Heap_stats>65% free 3571K/9991K
  堆的可用空间百分比与（活动对象数量）/（堆总大小）。
- 外部内存统计数据<External_memory_stats>external 4703K/5261K
  API 级别 10 及更低级别的外部分配内存（已分配内存量）/（发生回收的限值）。

- 暂停时间<Pause_time>paused 2ms+2ms
  堆越大，暂停时间越长。并发暂停时间显示了两个暂停：一个出现在回收开始时，另一个出现在回收快要完成时。
  在这些日志消息积聚时，请注意堆统计数据的增大（上面示例中的 3571K/9991K 值）。如果此值继续增大，可能会出现内存泄漏。

ART 日志消息

与 Dalvik 不同，ART 不会为未明确请求的垃圾回收记录消息。只有在认为垃圾回收速度较慢时才会打印垃圾回收。更确切地说，仅在垃圾回收暂停时间超过 5ms 或垃圾回收持续时间超过 100ms 时。如果应用未处于可察觉的暂停进程状态，那么其垃圾回收不会被视为较慢。始终会记录显式垃圾回收。

ART 会在其垃圾回收日志消息中包含以下信息：

```
I/art: <GC_Reason> <GC_Name> <Objects_freed>(<Size_freed>) AllocSpace
Objects, <Large_objects_freed>(<Large_object_size_freed>) <Heap_stats> LOS
objects, <Pause_time(s)>
示例:
I/art : Explicit concurrent mark sweep GC freed 104710(7MB) AllocSpace
objects, 21(416KB) LOS objects, 33% free, 25MB/38MB, paused 1.230ms total
67.216ms
```

- 垃圾回收原因<GC_Reason>Explicit
  什么触发了垃圾回收以及是哪种回收。可能出现的原因包括：
  - Concurrent
    不会暂停应用线程的并发垃圾回收。此垃圾回收在后台线程中运行，而且不会阻止分配。
  - Alloc
    您的应用在堆已满时尝试分配内存引起的垃圾回收。在这种情况下，分配线程中发生了垃圾回收。
  - Explicit
    由应用明确请求的垃圾回收，例如，通过调用 gc() 或 gc()。与 Dalvik 相同，在 ART 中，最佳做法是您应信任垃圾回收并避免请求显式垃圾回收（如果可能）。不建议使用显式垃圾回收，因为它们会阻止分配线程并不必要地浪费 CPU 周期。如果显式垃圾回收导致其他线程被抢占，那么它们也可能会导致卡顿（应用中出现间断、抖动或暂停）。
  - NativeAlloc
    原生分配（如位图或 RenderScript 分配对象）导致出现原生内存压力，进而引起的回收。
  - CollectorTransition
    由堆转换引起的回收；此回收由运行时切换垃圾回收引起。回收器转换包括将所有对象从空闲列表空间复制到碰撞指针空间（反之亦然）。当前，回收器转换仅在以下情况下出现：在 RAM 较小的设备上，应用将进程状态从可察觉的暂停状态变更为可察觉的非暂停状态（反之亦然）。
  - HomogeneousSpaceCompact
    齐性空间压缩是空闲列表空间到空闲列表空间压缩，通常在应用进入到可察觉的暂停进程状态时发生。这样做的主要原因是减少 RAM 使用量并对堆进行碎片整理。
  - DisableMovingGc
    这不是真正的垃圾回收原因，但请注意，发生并发堆压缩时，由于使用了 GetPrimitiveArrayCritical，回收遭到阻止。一般情况下，强烈建议不要使用 GetPrimitiveArrayCritical，因为它在移动回收器方面具有限制。
  - HeapTrim
    这不是垃圾回收原因，但请注意，堆修剪完成之前回收会一直受到阻止。
- 垃圾回收名称<GC_Name>concurrent mark sweep
  ART 具有可以运行的多种不同的垃圾回收。

    - Concurrent mark sweep (CMS)
      整个堆回收器，会释放和回收映像空间以外的所有其他空间。
    - Concurrent partial mark sweep
      几乎整个堆回收器，会回收除了映像空间和 zygote 空间以外的所有其他空间。
    - Concurrent sticky mark sweep
      生成回收器，只能释放自上次垃圾回收以来分配的对象。此垃圾回收比完整或部分标记清除运行得更频繁，因为它更快速且暂停时间更短。
    - Marksweep + semispace
      非并发、复制垃圾回收，用于堆转换以及齐性空间压缩（对堆进行碎片整理）。
- 释放的对象<Objects_freed>(<Size_freed>) GC freed 104710(7MB)
  此次垃圾回收从非大型对象空间回收的对象数量。
- 释放的大小AllocSpace Objects AllocSpace objects
  此次垃圾回收从非大型对象空间回收的字节数量。
- 释放的大型对象<Large_objects_freed>(<Large_object_size_freed>) 21(416KB) LOS objects
  此次垃圾回收从大型对象空间回收的对象数量。
- 释放的大型对象大小<Heap_stats> LOS objects 25MB/38MB,
  此次垃圾回收从大型对象空间回收的字节数量。
- 堆统计数据 33% free
  空闲百分比与（活动对象数量）/（堆总大小）。
- 暂停时间<Pause_time(s)>paused 1.230ms total 67.216ms
  通常情况下，暂停时间与垃圾回收运行时修改的对象引用数量成正比。当前，ART CMS 垃圾回收仅在垃圾回收即将完成时暂停一次。移动的垃圾回收暂停时间较长，会在大部分垃圾回收期间持续出现。
  如果您在 logcat 中看到大量的垃圾回收，请注意堆统计数据的增大（上面示例中的 25MB/38MB 值）。如果此值继续增大，且始终没有变小的趋势，则可能会出现内存泄漏。或者，如果您看到原因为“Alloc”的垃圾回收，那么您的操作已经快要达到堆容量，并且将很快出现 OOM 异常。

访问 Android Monitor

在连接的设备或模拟器上启动您的应用。
选择 View > Tool Windows > Android Monitor。
在 Android Monitor 的左上角，选择 Monitors 标签。
图 1. Android Monitor 及其三个监视器：Memory、CPU 和 GPU。在 Android Studio 中，垂直放大 Android Monitor 面板可以看到 Network 监视器。

捕捉堆转储

堆转储是应用堆中所有对象的快照。堆转储以一种名称为 HPROF 的二进制格式存储，您可以将其上传到分析工具（如 jhat）中。应用的堆转储包含应用堆整体状态的相关信息，以便您能够跟踪在查看堆更新时发现的问题。

在 Memory 监视器的顶部，点击 Dump Java Heap
Android Studio 会创建一个文件名为 application-id_yyyy.mm.dd_hh.mm.hprof 的堆快照文件，在 Android Studio 中打开文件，然后将文件添加到 Captures 标签的 Heap Snapshot 列表中。

在 Captures 标签中，右键点击文件，然后选择 Export to standard .hprof。
注：如果您需要更确切地了解转储的创建时间，可以通过调用 dumpHprofData() 在应用代码的关键点创建堆转储。

查看堆更新

使用 Android Monitor 在您与应用交互时查看应用堆的实时更新。实时更新提供了为不同应用操作分配的内存量的相关信息。您可以利用此信息确定是否任何操作占用了过多内存以及是否需要调整以减少占用的内存量。

与您的应用交互，在 Memory 监视器中，查看 Free 和 Alloated 内存。
点击 Dump Java Heap
在 Captures 标签中，双击堆快照文件以打开 HPROF 查看器。
要引起堆分配，请与您的应用交互，然后点击 Initiate GC
继续与您的应用交互，然后启动垃圾回收。观察每次垃圾回收的堆分配更新。确定应用中哪些操作导致过多分配，以及您可以从何处减少分配和释放资源。

分析堆转储

堆转储使用与 Java HPROF 工具中类似但不相同的格式提供。Android 堆转储的主要区别是在 Zygote 进程中进行了大量的分配。因为 Zygote 分配在所有应用进程之间分享，所以它们对您自己的堆分析影响不太大。

要分析堆转储，您可以使用标准工具，如 jhat。要使用 jhat，您需要将 HPROF 文件从 Android 格式转换为 Java SE HPROF 格式。要转换为 Java SE HPROF 格式，请使用 ANDROID_SDK/platform-tools/ 目录中提供的 hprof-conv 工具。运行包括两个参数的 hprof-conv 命令：原始 HPROF 文件和转换的 HPROF 文件的写入位置。例如：

```
hprof-conv heap-original.hprof heap-converted.hprof
```

您可以将转换的文件加载到可以识别 Java SE HPROF 格式的堆分析工具中。分析期间，请注意由下列任意情况引起的内存泄漏：

长时间引用 Activity、Context、View、Drawable 和其他对象，可能会保持对 Activity 或 Context 容器的引用。
可以保持 Activity 实例的非静态内部类，如 Runnable。
对象保持时间比所需时间长的缓存。

跟踪内存分配

跟踪内存分配可以让您更好地了解分配占用内存的对象的位置。您可以使用分配跟踪器查看特定的内存使用以及分析应用中的关键代码路径，如滚动。

例如，您可以使用分配跟踪器在应用中滑动列表时跟踪分配。跟踪让您可以看到滑动列表所需的所有内存分配，内存分配位于哪些线程上，以及内存分配来自何处。此类信息可以帮助您简化执行路径以减少执行的工作，从而改进应用的整体操作及其界面。

尽管不必要甚至也不可能将所有内存分配从您的性能关键型代码路径中移除，分配跟踪器仍可以帮助您识别代码中的重要问题。例如，应用可以在每次绘制时创建一个新的 Paint 对象。将 Paint 对象全局化是一个有助于提高性能的简单解决方法。

在连接的设备或模拟器上启动您的应用。
在 Android Studio 中，选择 View > Tool Windows > Android Monitor。
在 Android Monitor 的左上角，选择 Monitors 标签。
在内存监视器工具栏中，点击“Allocation Tracker”开始内存分配。
与您的应用交互。
再次点击“Allocation Tracker”停止分配跟踪。
Android Studio 会创建一个文件名为 application-id_yyyy.mm.dd_hh.mm.alloc 的分配文件，在 Android Studio 中打开该文件，然后将文件添加到 Captures 标签内的 Allocations 列表中。

在分配文件中，确定您的应用中哪些操作可能会引起过多分配，并确定应在应用中什么位置尝试减少分配和释放资源。
如需了解有关使用分配跟踪器的详细信息，请参阅分配跟踪器。

查看整体内存分配

为了进一步分析，您可能想要使用下面的 adb 命令观察应用内存在不同类型的 RAM 分配之间的划分情况：

`adb shell dumpsys meminfo <package_name|pid> [-d]`

-d 标志会打印与 Dalvik 和 ART 内存使用情况相关的更多信息。

输出列出了应用的所有当前分配，单位为千字节。

检查此信息时，您应熟悉下列类型的分配：

私有（干净和脏）RAM
这是仅由您的进程使用的内存。这是您的应用进程被破坏时系统可以回收的 RAM 量。通常情况下，最重要的部分是私有脏 RAM，它的开销最大，因为只有您的进程使用它，而且其内容仅存在于 RAM 中，所以无法被分页以进行存储（因为 Android 不使用交换）。所有的 Dalvik 和您进行的原生堆分配都将是私有脏 RAM；您与 Zygote 进程共享的 Dalvik 和原生分配是共享的脏 RAM。
按比例分配占用内存 (PSS)
这表示您的应用的 RAM 使用情况，考虑了在各进程之间共享 RAM 页的情况。您的进程独有的任何 RAM 页会直接影响其 PSS 值，而与其他进程共享的 RAM 页仅影响与共享量成比例的 PSS 值。例如，两个进程之间共享的 RAM 页会将其一半的大小贡献给每个进程的 PSS。
PSS 结果一个比较好的特性是，您可以将所有进程的 PSS 相加来确定所有进程正在使用的实际内存。这意味着 PSS 适合测定进程的实际 RAM 比重和比较其他进程的 RAM 使用情况与可用总 RAM。

例如，下面是 Nexus 5 设备上地图进程的输出。此处信息较多，但讨论的关键点如下所示。

`adb shell dumpsys meminfo com.google.android.apps.maps -d`
注：您看到的信息可能会与此处显示的信息稍有不同，因为输出的一些详细信息在不同平台版本之间会有所不同。

```
** MEMINFO in pid 18227 [com.google.android.apps.maps] **
                   Pss  Private  Private  Swapped     Heap     Heap     Heap
                 Total    Dirty    Clean    Dirty     Size    Alloc     Free
                ------   ------   ------   ------   ------   ------   ------
  Native Heap    10468    10408        0        0    20480    14462     6017
  Dalvik Heap    34340    33816        0        0    62436    53883     8553
 Dalvik Other      972      972        0        0
        Stack     1144     1144        0        0
      Gfx dev    35300    35300        0        0
    Other dev        5        0        4        0
     .so mmap     1943      504      188        0
    .apk mmap      598        0      136        0
    .ttf mmap      134        0       68        0
    .dex mmap     3908        0     3904        0
    .oat mmap     1344        0       56        0
    .art mmap     2037     1784       28        0
   Other mmap       30        4        0        0
   EGL mtrack    73072    73072        0        0
    GL mtrack    51044    51044        0        0
      Unknown      185      184        0        0
        TOTAL   216524   208232     4384        0    82916    68345    14570

 Dalvik Details
        .Heap     6568     6568        0        0
         .LOS    24771    24404        0        0
          .GC      500      500        0        0
```

```
    .JITCache      428      428        0        0
      .Zygote     1093      936        0        0
   .NonMoving     1908     1908        0        0
 .IndirectRef       44       44        0        0

 Objects
               Views:       90         ViewRootImpl:        1
         AppContexts:        4           Activities:        1
              Assets:        2        AssetManagers:        2
       Local Binders:       21        Proxy Binders:       28
       Parcel memory:       18         Parcel count:       74
    Death Recipients:        2      OpenSSL Sockets:        2
```

下面是 Gmail 应用的 Dalvik 上一个较旧版本的 dumpsys：

```
** MEMINFO in pid 9953 [com.google.android.gm] **
                   Pss     Pss  Shared Private  Shared Private    Heap    Heap    Heap
                 Total   Clean   Dirty   Dirty   Clean   Clean    Size   Alloc    Free
               ------  ------  ------  ------  ------  ------  ------  ------  ------
  Native Heap      0       0       0       0       0       0    7800    7637(6)  126
  Dalvik Heap   5110(3)    0    4136    4988(3)    0       0    9168    8958(6)  210
 Dalvik Other   2850       0    2684    2772       0       0
        Stack     36       0       8      36       0       0
       Cursor    136       0       0     136       0       0
       Ashmem     12       0      28       0       0       0
    Other dev    380       0      24     376       0       4
     .so mmap   5443(5) 1996    2584    2664(5) 5788    1996(5)
    .apk mmap    235      32       0       0    1252      32
    .ttf mmap     36      12       0       0      88      12
    .dex mmap   3019(5) 2148       0       0    8936    2148(5)
   Other mmap    107       0       8       8     324      68
      Unknown   6994(4)    0     252    6992(4)    0       0
        TOTAL  24358(1) 4188    9724   17972(2)16388    4260(2)16968   16595    336

 Objects
               Views:    426         ViewRootImpl:        3(8)
         AppContexts:      6(7)        Activities:        2(7)
              Assets:      2        AssetManagers:        2
       Local Binders:     64        Proxy Binders:       34
    Death Recipients:      0
     OpenSSL Sockets:      1

 SQL
         MEMORY_USED:   1739
  PAGECACHE_OVERFLOW:   1164          MALLOC_SIZE:       62
```

通常情况下，仅需关注 Pss Total 和 Private Dirty 列。一些情况下，Private Clean 和 Heap Alloc 列提供的数据也需要关注。您需要关注的不同内存分配（各行）的详细信息如下：

- Dalvik Heap
  您的应用中 Dalvik 分配占用的 RAM。Pss Total 包括所有 Zygote 分配（如上述 PSS 定义所述，通

过进程之间的共享内存量来衡量）。Private Dirty 数值是仅分配到您应用的堆的实际 RAM，由您自己的分配和任何 Zygote 分配页组成，这些分配页自从 Zygote 派生应用进程以来已被修改。
注：在包含 Dalvik Other 部分的更新的平台版本上，Dalvik 堆的 Pss Total 和 Private Dirty 数值不包括 Dalvik 开销（例如即时 (JIT) 编译和垃圾回收记录），而较旧的版本会在 Dalvik 中将其一并列出。

Heap Alloc 是 Dalvik 和原生堆分配器为您的应用跟踪的内存量。此值大于 Pss Total 和 Private Dirty，因为您的进程从 Zygote 派生，且包含您的进程与所有其他进程共享的分配。

- .so mmap 和 .dex mmap
  映射的 .so（原生）和 .dex（Dalvik 或 ART）代码占用的 RAM。Pss Total 数值包括应用之间共享的平台代码；Private Clean 是您的应用自己的代码。通常情况下，实际映射的内存更大 - 此处的 RAM 仅为应用执行的代码当前所需的 RAM。不过，.so mmap 具有较大的私有脏 RAM，因为在加载到其最终地址时对原生代码进行了修改。
- .oat mmap
  这是代码映像占用的 RAM 量，根据多个应用通常使用的预加载类计算。此映像在所有应用之间共享，不受特定应用影响。
- .art mmap
  这是堆映像占用的 RAM 量，根据多个应用通常使用的预加载类计算。此映像在所有应用之间共享，不受特定应用影响。尽管 ART 映像包含 Object 实例，它仍然不会计入您的堆大小。
- .Heap（仅带有 -d 标志）
  这是您的应用的堆内存量。不包括映像中的对象和大型对象空间，但包括 zygote 空间和非移动空间。
- .LOS（仅带有 -d 标志）
  这是由 ART 大型对象空间占用的 RAM 量。包括 zygote 大型对象。大型对象是所有大于 12KB 的原语数组分配。
- .GC（仅带有 -d 标志）
  这是内部垃圾回收量（考虑了应用开销）。真的没有任何办法减少这一开销。
- .JITCache（仅带有 -d 标志）
  这是 JIT 数据和代码缓存占用的内存量。通常情况下为 0，因为所有的应用都会在安装时编译。
- .Zygote（仅带有 -d 标志）
  这是 zygote 空间占用的内存量。zygote 空间在设备启动时创建且永远不会被分配。
- .NonMoving（仅带有 -d 标志）
  这是由 ART 非移动空间占用的 RAM 量。非移动空间包含特殊的不可移动对象，例如字段和方法。您可以通过在应用中使用更少的字段和方法来减少这一部分。
- .IndirectRef（仅带有 -d 标志）
  这是由 ART 间接引用表占用的 RAM 量。通常情况下，此量较小，但如果很高，可以通过减少使用的本地和全局 JNI 引用数量来减少此 RAM 量。
- Unknown
  系统无法将其分类到其他更具体的一个项中的任何 RAM 页。当前，此类 RAM 页主要包含原生分配，由于地址空间布局随机化 (ASLR) 而无法在收集此数据时通过工具识别。与 Dalvik 堆相同，Unknown 的 Pss Total 考虑了与 Zygote 的共享，且 Private Dirty 是仅由您的应用占有的未知 RAM。
- TOTAL
  您的进程占用的按比例分配占用内存 (PSS) 总量。等于上方所有 PSS 字段的总和。表示您的进程占用的内存量占整体内存的比重，可以直接与其他进程和可用总 RAM 比较。
  Private Dirty 和 Private Clean 是您的进程中的总分配，未与其他进程共享。它们（尤其是 Private Dirty）等于您的进程被破坏后将释放回系统中的 RAM 量。脏 RAM 是因为已被修改而必须保持在 RAM 中的 RAM 页（因为没有交换）；干净 RAM 是已从某个持久性文件（例如正在执行的代码）映射的 RAM 页，如果一段时间不用，可以移出分页。
- ViewRootImpl
  您的进程中当前活动的根视图数量。每个根视图都与一个窗口关联，因此有助于您确定涉及对话框或其他窗口的内存泄漏。

- AppContexts 和 Activities
  您的进程中当前活动的应用 Context 和 Activity 对象数量。这可以帮助您快速确定由于存在静态引用（比较常见）而无法进行垃圾回收的已泄漏 Activity 对象。这些对象经常拥有很多关联的其他分配，因此成为跟踪大型内存泄漏的一种不错的方式。
  注：View 或 Drawable 对象也会保持对其源 Activity 的引用，因此保持 View 或 Drawable 对象也会导致您的应用泄漏 Activity。

触发内存泄漏

使用上面介绍的工具时，您应积极地对自己的应用代码进行测试并尝试强制内存泄漏。在应用中引发内存泄漏的一种方式是，先让其运行一段时间，然后再检查堆。泄漏在堆中将逐渐汇聚到分配顶部。不过，泄漏越小，您越需要运行更长时间的应用才能看到泄漏。

您还可以通过以下方式之一触发内存泄漏：

- 将设备从纵向旋转为横向，然后在不同的活动状态下反复操作多次。旋转设备经常会使应用泄漏 Activity、Context 或 View 对象，因为系统会重新创建 Activity，而如果您的应用在其他地方保持对这些对象其中一个的引用，系统将无法对其进行垃圾回收。
- 处于不同的活动状态时，在您的应用与另一个应用之间切换（导航到主屏幕，然后返回到您的应用）。
  提示：您还可以使用 Monkey 测试框架执行上述步骤。如需了解有关运行 Monkey 测试框架的详细信息，请阅读 monkeyrunner 文档。